Articles » Internet of Things » Boards / Embedded devices » General

Fast digital I/O for Arduino

Jan Dolinay, 9 Mar 2015

4.89 (21 votes)

This article describes faster but still easy-to-use version of digital I/O for Arduino

Download source code and examples - dio2.zip - 26.9 KB

Introduction

This article describes my version of digital input/output functions for Arduino, which work faster than the 'built-in' functions while remaining as easy to use and portable as the original ones. If you just want to try the new functions, feel free to go directly to "Using the code" section, otherwise, read on for some introduction to the problem.

It is rather well known that the functions for digital I/O in Arduino are quite slow. It takes about 4 microseconds to change the logical level of an output pin (for example, to turn on an LED) using the Arduino `digitalWrite` function, while it takes less than 0.1 microsecond if you write the code 'natively', using the I/O registers of the Atmel AVR microcontroller (which is the brain of the Arduino board). Sure, for most users it does not matter whether it takes 0.1 or 4 microseconds, it is still fast enough, but there are situations where the speed or power consumption are critical.

Some time ago I wrote an article about this topic - 'Why is the digital I/O in Arduino slow and what can be done about it?'. This article was mostly about why the functions are slow; it did not offer any easy-to-use solution. In this new article I finally present my solution.

Certainly, there is the ultimate solution - use the I/O registers directly. In other words, get closer to the hardware. But by this you lose the portability of your program. While `digitalWrite` will work with any Arduino board, this approach will only work with one board and you will have to modify the code to use it with different board (e.g. if you move from Arduino Uno to Arduino Mega). Moreover, you need to understand the I/O registers to be able to use them.

I wondered if the digital I/O could be made faster, but also portable and easy to use. I tried various options (described at the end of this article) and came up with the solution presented here.

Main features of the new digital I/O implementation

- Implemented and tested for standard Arduino (Uno) and Arduino Mega.
- Easy to port to other boards.
- Very fast if pin number is a constant, but still considerably faster than standard Arduino I/O when pin number is a variable.
- Can be used in the same way as the Arduino functions, just add '2' to the function name, e.g. `digitalWrite2(13, HIGH);`.
- If you do not mind using special pin codes instead of simple pin numbers, you can use even faster functions, e.g. `digitalWrite2f(DP1, HIGH);`.

- Easy to add to Arduino by copying just 3 files.
- Easy way to decide if you prefer speed or minimal size of your program.
- Proper handling of pin number which is out of range (unlike in standard Arduino I/O).

I know it sounds like an advertisement, so here are the numbers. The following tables compare the speed of the standard I/O in Arduino and the new version, which I call **I/O 2** for short (and for lack of imagination).

Note: '*us*' is used as a symbol for microsecond.

Arduino Standard (Uno)

	pin is constant	pin is variable	test program size
Arduino I/O "as is"	4.1 us	4.1 us	3098 B
Arduino I/O without timer check (see note 1 below)	3.4 us	3.4 us	2998 B
I/O 2 with integer arguments (see note 2 below)	0.8 us	2.0 us (2.8 us)	2876 B (2944 B)
I/O 2 with its native arguments (pin code)	0.6 us	1.1 us (1.9 us)	2856 B (2924 B)

Arduino Mega

	pin is constant	pin is variable
Arduino I/O "as is"	6.8 us	6.8 us
Arduino I/O without timer check (see note 1 below)	3.4 us	3.4 us
I/O 2 with integer arguments (see note 2 below)	1.0 us	2.2 us (3.2 us)
I/O 2 with its native arguments (pin code)	0.8 us	1.4 us (2.3 us)

The results were obtained using Arduino software version 1.0.5-r2; test programs were build in the Arduino IDE with default settings.

The numbers in parentheses for I/O 2 functions are times obtained with user option set to prefer small size of the program rather than speed (which in fact means the I/O functions are not 'inlined 'into the code but called).

Note 1: In the Arduino implementation of `digitalWrite` and `digitalRead` there is a check whether the affected pin is used by a timer. It seems 'silly' to do this each time a pin is written or read - as the Arduino developer(s) themselves write in a source comment - so I did not include this check in my digital I/O functions. To provide fair comparison I measured the speed of the standard Arduino functions with this check disabled besides the standard 'out-of-the-box' version. Notice that for Arduino Mega this 'silly' timer check accounts for about half of the time it takes to execute `digitalWrite`!

Note 2: For the digital I/O functions I created, the 'native' identification of a pin is a 'pin code', rather than plain pin number used in Arduino. However, I realize that for many people it is easier to use plain numbers in their programs, so I created wrapper functions which take the pin number as an integer, convert it to the pin code and then call the native function. To my surprise (and satisfaction) even those wrapper functions are still quite a bit faster than the original Arduino I/O.

New speed results with Arduino IDE 1.6.0

In March 2015 I measured the speed again using the new Arduino 1.6.0 IDE and different, more exact method: switching output pin HIGH and LOW at full speed and measuring the output with oscilloscope. Here are the results for Arduino Uno:

digitalWrite speed

Function	Speed in microseconds

Arduino I/O "as is"	5.18
Arduino I/O without timer check	3.52
DIO2 with integer argument, non-constant	1.79
DIO2 with integer argument, constant	1.16
DIO2 with native argument, non-constant	0.97
DIO2 with native argument, constant	0.19

There are two big differences compared to the older results:

- Arduino "As is" speed is lower than it was (5.2 vs. 4.1 us). I guess it it because some changes in the code which checks whether the pin is on timer, because the speed without this check is about the same.
- The DIO2 speed with native argument (pin code) as a constant is much higher. Here I guess I made a mistake when measuring the speed earlier. The 0.19 us is pretty close to the expected result, which is 0.125 us (single SBI instruction which takes 2 CPU clock cycles at 16 MHz). If you run the test program included in the library, which measures 100 digitalWrites, you obtain about 0.35 us, because of the loop overhead.

Using the code

There are two ways how to use the I/O 2 functions in your program:
1) Install as Arduino library (named DIO2)
OR
2) Copy 3 files into your Arduino installation

Option 2 was used from the beginning and is still supported. I added the library option (1) in March 2015 as this seems to be the right way to extend the Arduino environment. Unfortunately, you still need to copy one file into the right folder in your Arduino location, because it is not possible to determine which board is selected from within the library code and there needs to be board-specific definition of the pins. The advantage of the library option is that you have syntax coloring in the editor for the new functions, easy access to example programs and if you need to build your program for a different board than the Uno or Mega supported in DIO2 now, you will not get errors.

Option 1 - Use as Arduino library

Step 1: Install the DIO2 library as any other Arduino library, that is extract the downloaded files into you Arduino libraries folder or use the automatic library install from the Arduino drop-down menu Sketch > Import Library.
Note that the automatic library install will install the library into your user profile only; into the Documents\Arduino\libraries folder (at least this is where I found it on my Win7 computer).

Step 2: Copy the **pin2_arduino.h** file for the board(s) you plan to use into the appropriate folder in your Arduino location.
You will find this file in the attached zip file (or in the extracted library folder) in [zip file]**\board\[board]**, where [board] is *standard* for Arduino Uno and *mega* for Arduino Mega.
This destination folder is:

For Arduino 1.6.0 IDE:
[your_arduino_location]\hardware\arduino\avr\variants\[board]
Examples: c:\arduino-1.6.0\hardware\arduino\avr\variants\standard or
c:\arduino-1.6.0\hardware\arduino\avr\variants\mega

For the older Arduino 1.0.x IDE:
[your_arduino_location]\hardware\arduino\variants\[board]
Examples: c:\arduino-1.0.5-r2\hardware\arduino\variants\standard or
c:\arduino-1.0.5-r2\hardware\arduino\variants\mega.

Please note that the pin2_arduino.h file is different for Arduino standard and Arduino Mega. Use the appropriate file for your Arduino variant.

Also note that you are not overwriting anything in your Arduino installation and you can still use the original digital I/O functions.

Option 2 - Copy required files to your Arduino location

If you decide to use this option rather than library, you need to copy 3 files into appropriate folders in your Arduino location. You will find these files in the attached zip file; the source and destination locations are as follows:

Arduino 1.6.0 IDE
Copy **arduino2.h** and **digital2.c** from **[zip file]\src** to **[your_arduino_location]\hardware\arduino\avr\cores\arduino\**.

Copy **pins2_arduino.h** from **[zip file]\board\standard** or **mega** to **[your_arduino_location]\hardware\arduino\avr\variants \standard** or **mega**.

Arduino 1.0.x IDE
Copy **arduino2.h** and **digital2.c** from **[zip file]\src** to **[your_arduino_location]\hardware\arduino\cores\arduino\**

Copy **pins2_arduino.h** from **[zip file]\board\standard** or **mega** to **[your_arduino_location]\hardware\arduino\variants \standard** or **mega**.

Note that using this option 2 has one disadvantage: if you need to build a program for other Arduino variant than Uno or Mega, you will encounter build error because of missing pins2_arduino.h file for this variant. To solve this, you can copy the "dummy" pins2_arduino.h file provided in [zip file]\board\dummy to the appropriate folder in the Arduino variants folder.

Test program

Once the above is done, start Arduino IDE and create a new program (sketch) as you normally do. You can now use the I/O 2 functions described below instead of the standard Arduino functions.

Here is an example program which blinks the LED at pin 13. It should work without change both for Arduino Uno and Mega.

```
/*
  Blink2 - Blink example modified for using digital I/O 2 instead of standard Arduino digital I/O.
 */
//include the fast I/O 2 functions
#include "arduino2.h"

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
const int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode2(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite2(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite2(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

There are only two small changes compared to the normal Arduino Blink example:

1. `#include "arduino2.h"` at the beginning of the file
2. use functions with the standard Arduino name appended with '2', e.g. `pinMode2()` instead of `pinMode()`.

There is also optional third change you could make - adding one #define above the #include "arduino2.h" line, which decides, whether you prefer fast program execution or smaller size:

```
#define  GPIO2_PREFER_SPEED    1
```

```
#include "arduino2.h"
```

If the `GPIO2_PREFER_SPEED` is set to 1, the I/O 2 functions will be declared `inline` which results in faster execution, but the program may grow big if you use the functions many times, because each time you call the function in your program, this call is replaced with the complete code of the function.

If `GPIO2_PREFER_SPEED` is 0, then the functions will be normally called, which means only one copy of each function exists in the program. This may save some program memory but the program runs slower (the function call takes some CPU time).

If you do not define this constant, default value 1 will be used. In most cases I recommend using value 1, that is, prefer the speed. Only if you would be running our of program memory, try to change the value to 0. However, it seems unlikely that some program would use up all the program memory just by digital I/O, so you should not really need to use the value 0.

If you want your digital I/O to run faster and can live with referring to a pin as `DP1` instead of 1, you can use functions with '2f' in the name. As mentioned above, these are the native functions of the I/O 2 library, which do not have the overhead of converting the integer pin number to special pin code. Here is the code for the Blink example using these native functions:

```
#include "arduino2.h" // include the fast I/O 2 functions
// The I/O 2 functions use special data type for pin
// Pin codes, such as DP13 are defined in pins2_arduino.h
const GPIO_pin_t led_pin = DP13;
void setup() {
 pinMode2f(led_pin, OUTPUT);
}

void loop() {
 digitalWrite2f(led_pin, HIGH);
 delay(1000);
 digitalWrite2f(led_pin, LOW);
 delay(1000);
}
```

Note that instead of `int` we define the pin as `GPIO_pin_t`. This data type is described in the Reference section below.

Example programs

In the attached file there are also example programs (sketches) for Arduino standard (Uno) and Mega. Three programs are provided for each board besides the simple blink examples metioned above:

- program for measuring the speed of digitalWrite
- program for testing all the outputs
- program for testing all the inputs

You will find instructions on use of these examples directly in the source code in comments.

Reference

`GPIO_pin_t` - data type used for digital pins. The name of a pin is **DP + pin number**. For example, the digital pin 13, where LED is connected, is referred to as `DP13`. Using this special data type instead of just simple number (int) allows the functions to work faster. See the 'How it works' section if you wonder how.

Functions which take simple pin number

These functions are fully compatible with the original Arduino I/O functions. Just add '2' to the name of the original function and you are using the new faster version.

`void <span style="font-weight: bold;">pinMode2</span>(uint8_t pin, uint8_t mode);`Set the direction of the pin. Possible options are INPUT, INPUT_PULLUP and OUTPUT.

`uint8_t <span style="font-weight: bold;">digitalRead2</span>(uint8_t pin);`Read the value at given pin. The pin should be previously configured as input with pinMode2. Return value is either HIGH or LOW.

`void <span style="font-weight: bold;">digitalWrite2</span>(uint8_t pin, uint8_t value);`Set given pin to HIGH or LOW level. The pin should be configured as output with pinMode2.

Functions which use pin code

These functions are faster than the previous ones. They differ from the original Arduino functions by the type of pin parameter - they use `GPIO_pin_t` instead of `uint8_t` (or `int`). So in your program, you define the pin as
`GPIO_pin_t pin = DP1;`
instead of
`int pin = 1;`.
The other parameters are the same as for standard Arduino functions.

void **pinMode2f**(GPIO_pin_t pin, uint8_t mode);
uint8_t **digitalRead2f**(GPIO_pin_t pin);
void **digitalWrite2f**(GPIO_pin_t pin, uint8_t value);

How it works

In this section I will describe how the new I/O functions work internally. This version of the functions is the result of many experiments with different approaches. For those interested, I describe those various approaches at the end of this article.

First, it is necessary to understand a little bit about how the digital I/O works in a microcontroller.

Simple explanation

This explanation is (hopefully) easier to understand, but also simplified and inaccurate. Imagine that for the microcontroller used in Arduino, the pin named 13 on the Arduino board would be known as a pin number 8229. It would be impractical for us to write 8229 in the program to refer to this pin. We want to be able to write simply 13. On the other hand, if we force the microcontroller to 'translate' the number 13 into 8229 each time we perform a read or write on the pin, this will cost some CPU time. If we do this translation 'off line' in our code by defining symbolic names for the pins, something like:
`#define PIN_13 8229`it will work faster.
This way, in our program, we can use the symbol `PIN_13`, which is still meaningful to a human, and in fact give the microcontroller the number it needs. The compiler will do the translation of PIN_13 into 8229.

More exact explanation

The brain of the Arduino, a microcontroller (MCU for short) organizes I/O pins into *ports*. Ports are named A, B, C, etc. Each port has 8 pins. So there are, for example, pins 0 to 7 on port A (referred to as PA0 to PA7), pins 0 to 7 on port B (PB0 to PB7), etc.

Each port is controlled by a set of *registers*. For example, there is a register for setting the direction (input or output) and a register for setting the voltage level at the pin (high or low). For now, let's just think about the register which sets the voltage level on an output pin. We call this register the *data register*. You can imagine this register as a normal variable in your program, which is 8 bits long. Each bit in this variable controls one pin of a given port. For example, to set pin number 0 in port D to logical 1 (high voltage) you need to set bit 0 in *data register* of port D (PORTD). The code in C can look like this:

```
PORTD  =  PORTD | 1;
```

or for short:

```
PORTD  |=  1;
```

You could also write it like this:

```
PORTD = 1;
```

But this way you would not only set the bit 0 to logical 1, but also set all the other pins to logical 0. You would set the value for the whole byte (all 8 bits/pins), and since you are writing value 1 (0000 0001 in binary), you would be actually saying 'set the first bit to logical 1 and all other bits to logical 0'. Usually, you want to change just one bit and leave the others untouched, that is why there is the OR operation.

The problem we are solving

In Arduino programs, we do not want to deal with the registers and ORs and ANDs, we just want to write `digitalWrite(pin, value);` and have the software library translate this into the appropriate `PORTx |= bit_mask;` The problem is how to do this translation as fast and possible and/or with as little code as possible.

The solution

After trying other options I came to the conclusion that the fastest operation of the I/O functions can be obtained if they get the information they need as their input parameter. This information is the address of the register, which controls the port and the bit mask of the pin within this port. These two pieces of information are encoded into a single 16-bit number, which I call *pin code*. The lower byte of this 16-bit number contains the address of the data register and the upper byte is the bit mask of the pin. Let's look at an example to make it clear:

For Arduino Uno an LED is connected to digital pin 13. Physically, in the MCU, this is pin 5 on port B (PB5). The address of the data register for port B is **0x25** (this information is obtained from the datasheet of the MCU). The bit mask for pin 5 is a byte, in which bit 5 has value 1, all other bits are 0. So it will be 00100000 in binary; **0x20** in hexadecimal. You can also create this number by writing (1 << 5) in a C program - shift 1 five times to the left. Note that bit 5 means actually the 6th bit from the right-hand side, because the bit numbers start from 0.

Put together, the pin code for Arduino pin 13 will look like this: **0x2025**.

It would be possible to work with the pin codes in this way, but the C language offers some tools to make the use more comfortable and safe. First, we will not define the pin code as a simple integer, but will create a new data type for it. This type is called `GPIO_pin_t` and means 'this is a pin', not just 'any integer'. If we do this, the compiler can help us detect incorrect use of our I/O functions. If someone by mistake calls our function which expects pin code with plain pin number, for example, `digitalWrite2f(13, HIGH);` (ERROR!) instead of `digitalWrite2f(DP13, HIGH);`, the compiler will complain "invalid conversion from 'int' to 'GPIO_pin_t'". Cool, isn't it? To achieve this, the pin codes are defined in an `enum` rather than using `#define`.

The 2nd trick is invisible for the 'end user' but helps a lot when defining the pins for a new board. It is a macro `GPIO_MAKE_PINCODE(port, pin)`. Thanks to this macro, the definition of the pin for given arduino board can look like this:

```
enum GPIO_pin_enum
{
 DP_INVALID = 0x0025,
 DP0 = GPIO_MAKE_PINCODE(MYPORTD,0),
 DP1 = GPIO_MAKE_PINCODE(MYPORTD,1),
 DP2 = GPIO_MAKE_PINCODE(MYPORTD,2),
...
```

which is much more readable than this:

```
enum GPIO_pin_enum
{
 DP_INVALID = 0x0025,
 DP0 = 0x012B,
 DP1 = 0x022B,
 DP2 = 0x042B,
 DP3 = 0x082B,
...
```

You can see the actual pin definitions for Arduino Uno and Mega in *pins2_arduino.h* files in the attached source code.

Compatibility with standard Arduino I/O

Personally, I think using a symbolic name for something like a pin comes as a natural thing when you deal with programming for some time. It is a pity that it is not used in Arduino, because now those millions of Arduino users would be used to writing something like `digitalWrite(PIN_13, HIGH);` instead of `digitalWrite(13, HIGH);` and once you have the actual pin 'hidden' behind some symbolic name, you can do some things with it - such as encoding useful information into it.

But since people got used to plain numbers for pins in Arduino, the I/O 2 library provides functions which take such numbers. If you use these functions, you do not have to change anything in your programs, just call `digitalWrite2` instead of `digitalWrite`.

How it works? There is an array defined in pins2_arduino.h file which contains the pin codes for pins available in given board. When you call e.g. `digitalWrite2`, it will convert the pin number into its pin code using this array and then call the native `digitalWrite2f` with this pin code. Of course, this conversion takes some time, but the functions are still faster than the original Arduino ones.

Integration into the Arduino package

I decided to organize the files for the new I/O in the same way they are organized in original Arduino. So there is *pins2_arduino.h* file in this folder:

[your_arduino_location]\hardware\arduino\variants\[variant_name], which contains the definitions specific for given Arduino variant (e.g. Uno or Mega).

And there is common code, placed in *arduino2.h* and *digital2.c* files located in:
[your_arduino_location]\hardware\arduino\cores\arduino\.

Porting the functions to another board

If you decide to use the I/O 2 functions on other Arduino board than the Uno or Mega, you will have to create your on version of the pin2_arduino.h file with the pin codes. This is relatively easy task. I would say easier than creating the various arrays needed for standard Arduino digital I/O. You just define the enum `GPIO_pin_enum` with pin codes appropriate for your board and the `gpio_pins_progmem` array, which maps these pin codes to the pin number. There should be no changes needed in the other parts of the code. For detailed instructions, please see the comments in the provided pins2_arduino.h files.

Points of Interest

Registers with address larger than one byte

For the MCU used in Arduino Mega, some I/O registers have address higher than 0xFF, which means their address will not fit into the single byte reserved for this purpose in the pin code. Luckily, their addresses are just between 0x0100 and 0x01F0, so we only need one extra bit to store such address. And luckily again, the registers with addresses which fit into one byte do not have addresses larger than about 0x40. So the top bit (bit 7) in the address byte is free to be used for this purpose. It requires some extra bit manipulation when working with the addresses, but it does not slow down the functions too much.

The pin is not an int

Please note that the use of `GPIO_pin_t` instead of `int` in this library does not limit your ability to store pins in a variable. You just have to write

```
GPIO_pin_t pin = DP1;
```

instead of

```
int pin = 1;
```

What it does affect, however, is your ability to easily manipulate pins in a loop. Say, you would need to switch on LEDs connected to pins 1 to 4 in a sequence. With standard Arduino `digitalWrite` you can write:

```
int pin;
for ( pin = 1; pin <= 4; pin++) {
 digitalWrite(pin, HIGH);
}
```

You **cannot** do this with the GPIO_pin_t because the numerical values of the pin codes are not related. If you do need/want to use loop for this and do want to use the faster native functions of I/O 2, you can use this trick:

```
GPIO_pin_t pins[] = {
 DP10,
 DP11,
 DP12,
 DP13,
 DP_INVALID,
};
...
int i;
for ( i = 0; pins[i] != DP_INVALID; i++) {
 digitalWrite2f(pins[i], HIGH);
}
```

Or, the easy way: use the Arduino-compatible functions which take integer pin number (e.g. digitalWrite2).

Turning off the timer

As mentioned earlier, in my implementation of the digital I/O, I omitted the check, whether given pin is on a timer and eventually disabling the timer, as it is done in the standard Arduino implementation. I assume that this check is intended for forgetful users who want to read or write a pin which they used previously in the same program as a PWM output (analogWrite). But I did not actually follow the code into all details, so it is possible that it will be needed to add the check and/or turning off the timer somewhere, probably into `pinMode`. If you encounter a problem related to this, please let me know.

Possible mistakes in use of the I/O 2 functions

It is not possible to mistakenly call `digitalWrite2f` with an integer instead of pin code; the compiler will report an error: "invalid conversion from 'int' to 'GPIO_pin_t'". It is, however, possible to mistakenly call `digitalWrite2` with a pin code instead of integer pin number. This will only result in a compiler warning and, unfortunately, the Arduino IDE does not show build warnings in default configuration. You may enable verbose output for compilation in the *File* > *Preferences* and check the output window in Arduino IDE for warnings, which are printed in red letters instead of while for the normal output. Note that there are some warnings in the Arduino library itself; this is not related to I/O 2.

Other methods for abstracting digital pins

In this section I will briefly describe the other methods I tried for making the Arduino I/O faster and of which the method described above came as a winner.

The basic presumption for all the methods is, that there must be simple interface for the user, such as `digitalWrite(pin, value)`. This leads to task of somehow converting the 'pin' parameter into appropriate port register and bit mask for the given pin and then reading/writing to the register using the bit mask.

If you want to do similar experiments, I can recommend using Atmel Studio for this. It contains simulator where you can measure the time it takes for your code to execute and step through it in the debugger, which makes the experiments much easier and faster. Later, for experiments with the hardware, you will probably want to use some real IDE. I used Eclipse with AVR plug-in, and only in the last phase, for final verification, returned to the Arduino IDE.

Option 1: Arrays in memory

This method is used in the digital I/O in Arduino. For detailed description of how it works please see my older article. I did not experiment with this approach on my own, but from measuring the speed of standard Arduino digitalWrite we can say it takes **about 3.4 us** for writing a pin (without checking the timer).

Option 2: Chain of conditions

This is the second method I described in the older article. It is used in Wiring, which is the ancestor of Arduino. In principle you compare the pin number with the ranges of values belonging to each port. For example, in Arduino Uno the pins 0 to 7 belong to port D, the pins 8 to 13 to port B, etc. The code could look like this:

```
if ( pin < 8)
 return &PORTD;
else if (pin < 14)
 return &PORTB;
else if ( pin < 20 )
 return &PORTC;
else
 return NOT_A_REG;
```

And the bit mask can be obtained from the pin number as follows:

```
/* the lower 3 bits are always the pin number if each port has 8 pins (0-7; 8-15;...)
 But Arduino standard has only 6 pins (0-5) for port B and C, so we have to compensate for the 2 
missing pins. */
#define GPIO_PIN_MASK(pin) ( (pin < 14) ? (1 << (pin & 0x07)) : (1 << ((pin+2) & 0x07) ) )
```

In the real implementation the code would be put info macro using the C Ternary conditional operator (`a ? b : c`) rather than if-else, but that it not important for our explanation here.

This method has one advantage compared to the option 1: if the pin number is a constant, the compiler can evaluate the conditions during compilation and the result is as efficient as if you used the registers directly and wrote something like PORTB |= 0x20; in your program. But if the pin is stored in a variable, the speed and size are similar to the option 1. To make things worse, the size and speed depend on how many pins there are on your board and how 'chaotic' is the mapping of these pins to the actual MCU ports. So while in case of Arduino Uno this method would be better than the option 1, thanks to the speed improvement for constant pins, for Arduino Mega it would probably perform much worse than the array version (with non-constant pins; with const pins it would still be very fast).

I tried this method in various combinations, such as obtaining both port and bit mask this way, or obtaining port and/or mask using a switch statement or an array, but the results were not satisfactory, over **2 us** for writing a digital pin for Arduino Uno. Given the expected (and considerable) decrease of speed with more pins (Arduino Mega), this is not a good option.

Option 3: Big switch

This method was the biggest surprise for me. We can write a switch which will directly contain the operation on the appropriate register with the appropriate mask for each possible pin number:

```
#define PORT_WRITE(port, pin, value) ((value == 0)?(port &= ~(1<<pin)):(port |= (1<<pin)) )

void switch_digitalWrite(uint8_t pin, uint8_t val)
{
    switch(pin)
    {
        case 0:
         PORT_WRITE(PORTD, 0, val);
         break;
        case 1:
         PORT_WRITE(PORTD, 1, val);
         break;
..
```

You will probably think that this is not a good approach. Maybe OK for Arduino Uno with 20 pins, but what about Arduino Mega with 70 pins? It will take forever to evaluate all the possible values if you, for example, want to use pin 60... Not so. The compiler is smart and 'optimizes' the switch statement into sort of table with jumps directly to the code for each case. Or, if you just assign some values to a variable in each case, it can even create an array in memory with the possible values for each case and simply read and store the value. As a result, the switch produces pretty fast code and the speed does not depend on the pin number. In my tests I could get **about 2.2 us** for pin write with switch and even below 2.0 us if I wrote the code with if statements in a sort of binary search.

Why I discarded this method was that you would need such a big switch for each register, or we can say for each operation - one for writing a pin, one for reading, one for setting the direction. Such code would be hard to port to other boards and to maintain - if you fix some bug in one of the switches, you have to remember to fix it also in all the others.

Option 4: Computing the port and mask from pin number

This method means that we somehow compute the register address and pin mask from the pin number. Pin number is still simple integer, as it is in Arduino. I will explain by one example, which I tried: Suppose we have Arduino Uno board with the Atmega328 MCU, but reorder the pins a little so that Arduino's digital pin 0 is port B, bit 0; digital pin 1 is port B, bit 1; and so on up to pin 7 which is port B, bit 7. Pin 8 is port C, bit 0; pin 9 is port C, bit 1;.... pin 16 is port D, bit 0, etc. So there is relationship between the pin number and port and bit. Then we can calculate the port number:

```
port_number  = pin / 8;
```

and bit number:

```
bit_number = pin % 8;
```

If the MCU registers are designed in a favorable way (as is the case with Atmega328), we just need to add some offset to the `port_number` to obtain the address of the register for this port. The offset would be 0x25 in our case, because port_number 0 means port B, which has the address 0x25. The bit mask is computed from bit number in a straightforward way by bit shift: (1 << bit_number).

This method looks promising, but there are drawbacks.
First, it would require renumbering the pins on the Arduino board - surely not feasible.
Or creating a conversion function, which would convert Arduino's pin number to our pin number - easy, but will cost CPU time.
Or including some conditions into the computation - also costs CPU time.

Secondly, the computations itself are not as fast as one would expect. For example, I was surprised how much time a simple bit shift (1 << N) takes if N is a variable and the compiler will actually have to generate a loop which shifts the bit N-times. My results for this methods were **about 2.0 us** in the ideal case, with reordered pins. Add the need to map the Arduino pin numbers to these reordered pin numbers and it is not as fast as you would want.

Option 5: Encoding port and mask into the pin definition

This is the option which won in my tests; it provided the best trade-off between speed and maintainability (and portability) of the code. The particular implementation I used is described in detail above. In general, we can say that the principle is to include the port address and bit mask in the definition of the pin. So the functions for digital I/O do not receive directly the number of the pin, but some kind of a code (or data structure) which includes all the information needed to manipulate the pin. The advantage compared to the other methods is, that it does not need 'expensive' computations, like shifting bits by variable number, or comparisons and branches.

History

2014-02-25 First version.

2014-05-30 Minor updates in the text; fixed version of the code uploaded which works also on Linux.

2015-02-27 Code updated to build in Arduino IDE 1.6.0, locations for files updated in text

2015-03-09 Changes to allow use as Arduino library (directories reorganized) and zip file renamed to dio2.zip. Also new speed tests added using oscilloscope (without error from loop control code).

License

Share

About the Author

Jan Dolinay

Tomas Bata University in Zlin

Czech Republic

Works at Tomas Bata University in Zlin, Czech Republic. Interested in programming in general and especially programming microcontrollers.

You may also be interested in...

Pro Keeping Up With PHP

10 Ways to Boost COBOL Application Development

Why is the digital I/O in Arduino slow and what can be done about it?

Intel® Quark™ SE Microcontroller C1000 Developer Kit - Accelerometer Tutorial

Arduino Tips & Tricks

Visual COBOL New Release: Small point. Big deal

Comments and Discussions

60 messages have been posted for this article Visit **https://www.codeproject.com/Articles/732646/Fast-digital-I-O-for-Arduino** to post and view comments on this article, or click **here** to get a print view with messages.